Interprocedural Analysis of Concurrent Programs Under a Context Bound*

Akash Lal[1]**, Tayssir Touili[2], Nicholas Kidd[1], and Thomas Reps[1,3]

[1] University of Wisconsin; Madison, WI; USA. {akash, kidd, reps}@cs.wisc.edu
[2] LIAFA; CNRS & University of Paris 7; Paris; France. touili@liafa.jussieu.fr
[3] GrammaTech, Inc.; Ithaca, NY; USA.

Abstract. Analysis of recursive programs in the presence of concurrency and shared memory is undecidable. In previous work, Qadeer and Rehof [23] showed that context-bounded analysis is decidable for recursive programs under a finite-state abstraction of program data. In this paper, we show that context-bounded analysis is decidable for certain families of infinite-state abstractions, and also provide a new symbolic algorithm for the finite-state case.

1 Introduction

This paper considers the analysis of concurrent programs with shared-memory and interleaving semantics. Such an analysis for recursive programs is, in general, undecidable, even with a finite-state abstraction of data (e.g., Boolean Programs [1]). As a consequence, to deal with concurrency soundly (i.e., capture all concurrent behaviors), some analyses give up precise handling of procedure call/return semantics. Alternatively, tools use inlining to unfold multi-procedure programs into single-procedure ones. This approach cannot handle recursive programs, and can cause an exponential blowup in size for non-recursive ones.

A different way to sidestep the undecidability issue is to limit the amount of concurrency by bounding the number of *context switches*, where a context switch is defined as the transfer of control from one thread to another. Such an approach is not sound because it does not capture all of the behaviors of a program; however, it has proven to be useful in tools for bug-finding because many bugs can be found after a few context switches [24, 23, 20]. For example, KISS [24] is a verification tool that analyzes programs for only up to two context switches; it was able to find a number of bugs in device drivers. We call the analysis of recursive, concurrent programs under a context bound *context-bounded analysis* (CBA). CBA does not impose any bound on the execution length between context switches. Thus, even under a context bound, the analysis still has to consider the possibility that the next switch takes place in any one of the (possibly infinite) states that may be reached after a context switch. Because of this, CBA still considers an infinite number of interleavings.

* Supported by NSF under grants CCF-0540955 and CCF-0524051.
** Supported by a Microsoft Research Graduate Fellowship.

Qadeer and Rehof [23] showed that CBA is decidable for recursive programs under a finite-state abstraction of data. This paper shows that CBA is decidable for certain families of infinite-state abstractions, and also provides a new symbolic algorithm for the finite-state case. We give conditions on the abstractions under which CBA can be solved precisely, along with a new algorithm for CBA. In addition to the usual conditions required for precise interprocedural analysis of sequential programs, we require the existence of a *tensor product* (see §6). We show that these conditions are satisfied by a class of abstractions, thus giving precise algorithms for CBA with those abstractions. These include finite-state abstractions, such as the ones used for verification of Boolean programs, as well as infinite-state abstractions, such as affine-relation analysis [19].

Our results are achieved using techniques that are quite different from the ones used in the Qadeer and Rehof (QR) algorithm [23]. In particular, to explore all possible interleavings, the QR algorithm crucially relies on the finiteness of the data abstraction because it enumerates all reachable (abstract) data states at a context switch. Our algorithm is based on *weighted transducers* (weighted automata that read input and write output) and requires no such enumeration in the case of finite-state data abstractions, which also makes it capable of handling infinite-state abstractions.

The contributions of this paper can be summarized as follows:

- We give sufficient conditions under which CBA is decidable for infinite-state abstractions, along with an algorithm. Our result proves that CBA can be decided for affine-relation analysis, i.e., we can precisely find all affine relationships between program variables that hold in a concurrent program (under the context bound).
- We show that the reachability relation of a weighted pushdown system (WPDS) can be encoded using a weighted transducer (§5), which generalizes a previous result for (unweighted) PDSs [8]. We use WPDSs to model each thread of the concurrent program, and the transducers can be understood as summarizing the (sequential) execution of a thread.
- We give precise algorithms for composing weighted transducers (§6), when tensor products exist for the weights. This generalizes previous work on manipulating weighted automata and transducers [17, 18].

The remainder of the paper is organized as follows. §2 introduces some terminology and notation. §3 sketches an alternative to the QR algorithm for finite-state abstractions; the rest of the paper generalize the algorithm to infinite-state abstractions. §4 gives background on WPDSs. §5 gives an efficient construction of transducers for WPDSs. §6 shows how weighted transducers can be composed. §7 discusses related work. Additional material can be found in [16].

2 Terminology and Notation

A context-bounded analysis (CBA) considers a set of concurrent threads that communicate via global variables. Synchronization is easily implementable using

global variables as locks. Analysis of such models is undecidable [25], i.e., it is not possible, in general, to determine whether or not a given configuration is reachable. Let n be the number of threads and let $t_1, t_2, \cdots, t_n$ denote the threads. We do not consider dynamic creation of threads in our model. (Dynamic creation of up to n threads can be encoded in the model [23].)

Let G be the set of global states (valuations of global variables) and L_i be the set of local states of t_i. Then the state space of the entire program consists of the global state paired with local states of each of the threads, i.e., the set of states is $G \times L_1 \times \cdots \times L_n$. Let the transition relation of thread t_i, which is a relation on $G \times L_i$, be denoted by $\Rightarrow_{t_i}$. If $(g, l_i) \Rightarrow_{t_i} (g', l_i')$, the transition $(g, l_1, \cdots, l_i, \cdots, l_n) \Rightarrow^c_{t_i} (g', l_1 \cdots, l_i', \cdots, l_n)$ is a valid transition for the concurrent program.

The execution of a concurrent program proceeds in a sequence of *execution contexts*. In an execution context, one thread has control and it executes a finite number of steps. The execution context changes at a *context switch* and control is passed to a different thread. The CBA problem is to find the set of reachable states of the concurrent program under a bound on the number of context switches. Formally, let k be the bound on the number of context switches; thus, there are $k+1$ execution contexts. Let $\Rightarrow^c$ be $(\cup_{i=1}^{n}(\Rightarrow^c_{t_i})^*)$, the transition relation that describes the effect of one execution context; we wish to find the reachable states in the transition relation given by $(\Rightarrow^c)^{k+1}$.

Definition 1. *A **pushdown system** (PDS) is a triple $\mathcal{P} = (P, \Gamma, \Delta)$, where P is a finite set of states or control locations, Γ is a finite set of stack symbols, and $\Delta \subseteq P \times \Gamma \times P \times \Gamma^*$ is a finite set of rules. A **configuration** of $\mathcal{P}$ is a pair $\langle p, u \rangle$ where $p \in P$ and $u \in \Gamma^*$. A rule $r \in \Delta$ is written as $\langle p, \gamma \rangle \hookrightarrow \langle p', u \rangle$, where $p, p' \in P$, $\gamma \in \Gamma$ and $u \in \Gamma^*$. These rules define a transition relation $\Rightarrow$ on configurations of $\mathcal{P}$ as follows: If $r = \langle p, \gamma \rangle \hookrightarrow \langle p', u' \rangle$, then $\langle p, \gamma u \rangle \Rightarrow \langle p', u' u \rangle$ for all $u \in \Gamma^*$. The reflexive transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$. For a set of configurations C, we define $pre^*(C) = \{c' \mid \exists c \in C : c' \Rightarrow^* c\}$ and $post^*(C) = \{c' \mid \exists c \in C : c \Rightarrow^* c'\}$, which are just backward and forward reachability under the transition relation $\Rightarrow$.*

Without loss of generality, we restrict the pushdown rules to have at most two stack symbols on the right-hand side [28].

PDSs can encode recursive programs with a finite-state data abstraction [28]: the data values get tracked by the PDS state, and recursion gets handled by the PDS stack. In this case, a PDS configuration represents a program state (current data values and stack). For sequential programs, the problem of interest is to find the set of all reachable configurations, starting from a given set of configurations. This can then be used, for example, for assertion checking (i.e., determining if a given assertion can ever fail) or to find the set of all data values that may arise at a program point (for dataflow analysis). Because the number of configurations of a PDS is unbounded, it is useful to use finite automata to describe regular sets of configurations.

Definition 2. *If $\mathcal{P} = (P, \Gamma, \Delta)$ is a PDS then a* $\mathcal{P}$**-automaton** *is a finite automaton $(Q, \Gamma, \rightarrow, P, F)$, where $Q \supseteq P$ is a finite set of states, $\rightarrow \subseteq Q \times \Gamma \times Q$ is the transition relation, P is the set of initial states, and F is the set of final states. We say that a configuration $\langle p, u \rangle$ is accepted by a $\mathcal{P}$-automaton if it can accept u when started in the state p. A set of configurations is* ***regular*** *if it is the language of some $\mathcal{P}$-automaton.*

For a regular set of configurations C, both $post^*(C)$ and $pre^*(C)$ are also regular [2, 7, 11]. The algorithms for computing $post^*$ and pre^*, called *poststar* and *prestar*, respectively, take a $\mathcal{P}$-automaton $\mathcal{A}$ as input, and if C is the set of configurations accepted by $\mathcal{A}$, produce $\mathcal{P}$-automata $\mathcal{A}_{post^*}$ and $\mathcal{A}_{pre^*}$ that accept the sets $post^*(C)$ and $pre^*(C)$, respectively [2, 10, 11].

3 A New Approach Using Thread Summarization

Between consecutive context switches only one thread is executing, and a concurrent program acts like a sequential program. However, a recursive thread can reach an infinite number of states before the next context switch, because it has an unbounded stack. A CBA must consider the possibility that a context switch occurs at any of these states.

The QR algorithm works under the assumption that the set G is finite. It uses a PDS to encode each thread (using a finite-state data abstraction). The algorithm follows a computation tree, each node of which represents a set of states $\{g\} \times S_1 \times \cdots S_n$ of the concurrent program, where $S_i \subseteq L_i$, i.e., the set of states represented by each node has the same global state. The root of the tree represents the set of initial states, and nodes at the i^{th} level represent all states reachable after $i - 1$ context switches.

At each node, including the root, the computation tree branches: For each choice of thread t_j that receives control, a new set of states is obtained by running *poststar* on the PDS for thread t_j, starting with the node's (unique) global state and local states of t_j. The local states of t_h, $h \neq j$, are held fixed. The resulting set is split according to the reachable global states, to create multiple new nodes (maintaining the invariant that each node only represents a single global state). This process is repeated until the computation tree is completely built for $k + 1$ levels, giving all reachable states in k or fewer context switches. The "splitting" of nodes depending on reachable global states causes branching of the computation tree proportional to the size of G at each level.

Another drawback of the QR algorithm is its use of *poststar* to compute the forward reachable states of a given thread. *Poststar* represents a function that maps a starting set of configurations to a set of reachable configurations. Hence, *poststar* needs to be re-executed if the starting set changes. The discovery of new starting sets forces the QR algorithm to make multiple calls on *poststar* (for a given thread) to compute different sets of forward reachable states.

A similar problem arises in interprocedural analysis of sequential programs: a procedure can be called from multiple places with multiple different input

values. Instead of reanalyzing a procedure for each input value, a more efficient approach is to analyze each procedure independently of the calling context to create a *summary*. The summary describes the effect of executing the procedure in any calling context, in terms of a *relation* between inputs and outputs.

Our first step is to develop a new algorithm—based on a suitable form of summary relation for threads—for the case of unweighted PDSs. The motivation is to develop an algorithm that avoids enumerating all global states at a context switch, and then use that algorithm as the starting point for a generalization that handles infinite-state abstractions. A difficulty that we face, however, is that each summary relation must relate starting sets of configurations to reachable sets of configurations. Because both of these sets can be infinite, we need summary relations to be representable symbolically.

Our approach to generalizing the QR algorithm (for both finite-state and infinite-state data abstractions) is based on the following observation:

Observation 1 *One can construct an appropriate summary of a thread's behavior using a finite-state transducer (an automaton with input and output tapes).*

Definition 3. *A **finite-state transducer** τ is a tuple $(Q, \Sigma_i, \Sigma_o, \lambda, I, F)$, where Q is a finite set of states, Σ_i and Σ_o are input and output alphabets, $\lambda \subseteq Q \times (\Sigma_i \cup \{\varepsilon\}) \times (\Sigma_o \cup \{\varepsilon\}) \times Q$ is the transition relation, $I \subseteq Q$ is the set of initial states, and $F \subseteq Q$ is the set of final states. If $(q_1, a, b, q_2) \in \lambda$, written as $q_1 \xrightarrow{a/b} q_2$, we say that the transducer can go from state q_1 to q_2 on input a, and outputs the symbol b. For state $q \in I$, the transducer accepts string $\sigma_i \in \Sigma_i^*$ with output $\sigma_o \in \Sigma_o^*$ if there is a path from q to a final state that takes input σ_i and outputs σ_o. The **language** of the transducer $\mathcal{L}(\tau)$ is the relation $\{(\sigma_i, \sigma_o) \in \Sigma_i^* \times \Sigma_o^* \mid$ the transducer can output string σ_o when the input is $\sigma_i\}$.*

In the case of finite-state abstractions, and each thread represented as some PDS $\mathcal{P}$, one can construct a transducer $\tau_{\mathcal{P}}$ whose language equals $\Rightarrow^*$, the transitive closure of $\mathcal{P}$'s transition relation: The transducer accepts a pair (c_1, c_2) if a thread, when started in state c_1, can reach state c_2. The advantage of using transducers is that they are closed under relational composition:

Lemma 1. *Given transducers τ_1 and τ_2 with input and output alphabet Σ, one can construct a transducer $(\tau_1; \tau_2)$ such that $\mathcal{L}(\tau_1; \tau_2) = \mathcal{L}(\tau_1); \mathcal{L}(\tau_2)$, where the latter ";" denotes composition of relations. Similarly, if $\mathcal{A}$ is an automaton with alphabet Σ, one can construct an automaton $\tau_1(\mathcal{A})$ such that its language is the image of $\mathcal{L}(\mathcal{A})$ under $\mathcal{L}(\tau_1)$, i.e., the set $\{u \in \Sigma^* \mid \exists u' \in \mathcal{L}(\mathcal{A}), (u', u) \in \mathcal{L}(\tau_1)\}$.*

Both of these constructions are carried out in a manner similar to intersection of automata [13]. One can also take the union of transducers (union of their languages) in a manner similar to union of automata.

In the case of CBA with a finite-state data abstraction, each thread is represented using a PDS. We construct a transducer τ_{t_i} for the transition relation $\Rightarrow^*_{t_i}$. By extending τ_{t_i} to perform the identity transformation on stack symbols of threads other than t_i (using transitions of the form $p \xrightarrow{\gamma/\gamma} q$), we obtain

a transducer $\tau^c_{t_i}$ for $(\Rightarrow^c_{t_i})^*$. Next, a union of these transducers gives τ^c, which represents $\Rightarrow^c$. Performing the composition of τ^c k times with itself gives us a transducer τ that represents $(\Rightarrow^c)^{k+1}$. If an automaton $\mathcal{A}$ captures the set of starting states of the concurrent program, $\tau(\mathcal{A})$ gives a single automaton for the set of all reachable states in the program (under the context bound).

The rest of the paper generalizes this approach to infinite-state abstractions by going from PDSs to WPDSs. This requires the construction (§5) and composition (§6) of weighted transducers.

4 Weighted Pushdown Systems (WPDSs)

A WPDS is a PDS augmented with weights drawn from a *bounded idempotent semiring* [27, 4]. Such semirings are powerful enough to encode finite-state data abstractions, as used in bitvector dataflow analysis, Boolean program verification [1], and the IFDS dataflow-analysis framework [26], as well as infinite-state data abstractions, such as linear-constant propagation and affine-relation analysis [19]. We review some of this here; see also [27].

Weights encode the effect that each statement (or PDS rule) has on the data state of the program. They can be thought of as abstract transformers that specify how the abstract state changes when a statement is executed.

Definition 4. *A* **bounded idempotent semiring** *(or "weight domain") is a tuple $(D, \oplus, \otimes, \overline{0}, \overline{1})$, where D is a set of* **weights**, *$\overline{0}, \overline{1} \in D$, and $\oplus$ (combine) and $\otimes$ (extend) are binary operators on D such that*

1. *$(D, \oplus)$ is a commutative monoid with $\overline{0}$ as its neutral element, and where $\oplus$ is idempotent. $(D, \otimes)$ is a monoid with the neutral element $\overline{1}$.*
2. *$\otimes$ distributes over $\oplus$, i.e., for all $a, b, c \in D$ we have*
   $$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c) \text{ and } (a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c) .$$
3. *$\overline{0}$ is an annihilator with respect to $\otimes$, i.e., for all $a \in D$, $a \otimes \overline{0} = \overline{0} = \overline{0} \otimes a$.*
4. *In the partial order $\sqsubseteq$ defined by $\forall a, b \in D,\ a \sqsubseteq b$ iff $a \oplus b = a$, there are no infinite descending chains.*

The *height* of a weight domain is defined to be the length of the longest descending chain in the domain. In this paper, we assume the height to be bounded for ease of discussing complexity results, but WPDSs, and the algorithms in this paper, can also be used in certain cases when the height is unbounded (as long as there are no infinite descending chains). $O_s(.)$ denotes the time bound in terms of semiring operations.

Often, weights are data transformers: extend is composition; combine is *meet*; $\overline{0}$ is the transformer for an infeasible path; and $\overline{1}$ is the identity transformer.

Definition 5. *A* **weighted pushdown system** *is a triple $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ where $\mathcal{P} = (P, \Gamma, \Delta)$ is a pushdown system, $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ is a bounded idempotent semiring and $f : \Delta \to D$ is a map that assigns a weight to each rule of $\mathcal{P}$.*

Let $\sigma \in \Delta^*$ be a sequence of rules. Using f, we can associate a value to σ, i.e., if $\sigma = [r_1, \ldots, r_k]$, then we define $v(\sigma) \stackrel{\text{def}}{=} f(r_1) \otimes \ldots \otimes f(r_k)$. Moreover, for any two configurations c and c' of $\mathcal{P}$, we use $path(c, c')$ to denote the set of all rule sequences that transform c into c'. If $\sigma \in path(c, c')$, then we say $c \Rightarrow^\sigma c'$. Reachability problems on PDSs are generalized to WPDSs as follows:

Definition 6. *Let $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ be a WPDS, where $\mathcal{P} = (P, \Gamma, \Delta)$, and let $S, T \subseteq P \times \Gamma^*$ be regular sets of configurations. Then the* ***meet-over-all-valid-paths*** *value* $\mathrm{MOVP}(S, T)$ *is defined as* $\bigoplus\{v(\sigma) \mid s \Rightarrow^\sigma t, s \in S, t \in T\}$.

A PDS is simply a WPDS with the *Boolean weight domain* $(\{\overline{0}, \overline{1}\}, \oplus, \otimes, \overline{0}, \overline{1})$ and weight assignment $f(r) = \overline{1}$ for all rules $r \in \Delta$. In this case, $\mathrm{MOVP}(S, T) = \overline{1}$ iff there is a path from a configuration in S to a configuration in T, i.e., $post^*(S) \cap T$ and $S \cap pre^*(T)$ are non-empty.

One way of modeling programs as WPDSs is as follows: the PDS models the control flow of the program and the weight domain models transformers for an abstraction of the program's data. Examples are given in [16].

4.1 Solving for the MOVP Value

There are two algorithms for solving for MOVP values, called *prestar* and *poststar* (by analogy with the algorithms for PDSs) [27]. Their input is an automaton that accepts the set of initial configurations. Their output is a *weighted automaton*:

Definition 7. *Given a WPDS $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$, a* ***$\mathcal{W}$-automaton*** *$\mathcal{A}$ is a $\mathcal{P}$-automaton, where each transition in the automaton is labeled with a weight. The weight of a path in the automaton is obtained by taking an extend of the weights on the transitions in the path in either a forward or backward direction. The automaton is said to accept a configuration $c = \langle p, u \rangle$ with weight $w = \mathcal{A}(c)$ if w is the combine of weights of all accepting paths for u starting from state p in $\mathcal{A}$. We call the automaton a* ***backward $\mathcal{W}$-automaton*** *if the weight of a path is read backwards, and a* ***forward $\mathcal{W}$-automaton*** *otherwise.*

Let $\mathcal{A}$ be an unweighted automaton and $\mathcal{L}(\mathcal{A})$ be the set of configurations accepted by it. Then there is an algorithm *prestar*$(\mathcal{A})$ that produces a forward weighted automaton $\mathcal{A}_{pre^*}$ as output, such that $\mathcal{A}_{pre^*}(c) = \mathrm{MOVP}(\{c\}, \mathcal{L}(\mathcal{A}))$, and an algorithm *poststar*$(\mathcal{A})$ produces a backward weighted automaton $\mathcal{A}_{post^*}$ as output, such that $\mathcal{A}_{post^*}(c) = \mathrm{MOVP}(\mathcal{L}(\mathcal{A}), \{c\})$ [27]. For a weighted automaton $\mathcal{A}'$, we define $\mathcal{A}'(C) = \bigoplus\{\mathcal{A}'(c) \mid c \in C\}$. The values of $\mathcal{A}'(c)$ and $\mathcal{A}'(C)$ can be computed by the algorithms presented in [27].

Lemma 2. *[27] Given a WPDS with PDS $\mathcal{P} = (P, \Gamma, \Delta)$, if $\mathcal{A} = (Q, \Gamma, \rightarrow, P, F)$ is a $\mathcal{P}$-automaton, poststar$(\mathcal{A})$ produces an automaton with at most $|Q| + |\Delta|$ states and runs in time $O_s(|P||\Delta|(|Q_0| + |\Delta|)H + |P||\lambda_0|H)$, where $Q_0 = Q \backslash P$, $\lambda_0 \subseteq \rightarrow$ is the set of all transitions leading from states in Q_0, and H is the height of the weight domain.*

4.2 CBA Problem Definition

Definition 8. *A **weighted relation** on a set S, weighted with semiring $(D, \oplus, \otimes, \overline{0}, \overline{1})$, is a function from $(S \times S)$ to D. The composition of two weighted relations R_1 and R_2 is defined as $(R_1; R_2)(s_1, s_3) = \oplus\{w_1 \otimes w_2 \mid \exists s_2 \in S : w_1 = R_1(s_1, s_2), w_2 = R_2(s_2, s_3)\}$. The union of the two weighted relations is defined as $(R_1 \cup R_2)(s_1, s_2) = R_1(s_1, s_2) \oplus R_2(s_1, s_2)$. The identity relation is the function that maps each pair (s, s) to $\overline{1}$ and others to $\overline{0}$. The reflexive-transitive closure is defined in terms of these operations, as before.*

The transition relation of a WPDS is a weighted relation over the set of PDS configurations. For configurations c_1 and c_2, if $r_1, \cdots, r_m$ are all the rules such that $c_1 \Rightarrow^{r_i} c_2$, then $(c_1, c_2, \oplus_i f(r_i))$ is in the weighted relation of the WPDS. In a slight abuse of notation, we use $\Rightarrow$ and its variants for the weighted transition relation of a WPDS. Note that the weighted relation $\Rightarrow^*$ maps a configuration pair (c_1, c_2) to $\text{MOVP}(\{c_1\}, \{c_2\})$.

The CBA problem is defined as in §2, except that all relations are weighted. Each thread is modeled as a WPDS. Given the weighted relation $(\Rightarrow^c)^{k+1}$ as R, the set of initial states S and a set of final states T (of the concurrent program), we want to be able to compute the weight $R(S, T) = \oplus\{R(s, t) \mid s \in S, t \in T\}$. This captures the net transformation on the data state between S and T: it is the combine over the weights of all paths involving at most k context switches that go from a state in S to a state in T. Our results from §5 and §6 allow us to compute this value when S and T are regular.

5 Weighted Transducers

In this section, we show how to construct a weighted transducer for the weighted relation $\Rightarrow^*$ of a WPDS. We defer the definition of a weighted transducer to a little later in this section (Defn. 9). Our solution uses the following observation about paths in a PDS's transition relation. Every path $\sigma \in \Delta^*$ that starts from a configuration $\langle p_1, \gamma_1\gamma_2\cdots\gamma_n\rangle$ can be decomposed as $\sigma = \sigma_1\sigma_2\cdots\sigma_k\sigma_{k+1}$ (see Fig. 1) such that $\langle p_i, \gamma_i\rangle \Rightarrow^{\sigma_i} \langle p_{i+1}, \varepsilon\rangle$ for $1 \leq i \leq k$, and $\langle p_{k+1}, \gamma_{k+1}\rangle \Rightarrow^{\sigma_{k+1}} \langle p_{k+2}, u_1u_2\cdots u_j\rangle$: every path has zero or more *pop phases* $(\sigma_1, \sigma_2, \cdots, \sigma_k)$ followed by a single *growth phase* (σ_{k+1}):

1. **Pop-phase:** A path such that the net effect of the pushes and pops performed along the path is to take $\langle p, \gamma u\rangle$ to $\langle p', u\rangle$, without looking at $u \in \Gamma^*$. Equivalently, it can take $\langle p, \gamma\rangle$ to $\langle p', \varepsilon\rangle$.
2. **Growth-phase:** A path such that the net effect of the pushes and pops performed along the path is to take $\langle p, \gamma u\rangle$ to $\langle p', u'u\rangle$ with $u' \in \Gamma^+$, without looking at $u \in \Gamma^*$. Equivalently, it can take $\langle p, \gamma\rangle$ to $\langle p', u'\rangle$.

Intuitively, this holds because for a path to look at γ_2, it must pop off γ_1. If it does not pop off γ_1, then the path is in a growth phase starting from γ_1. Otherwise, the path just completed a pop phase. We construct the transducer for a WPDS by computing the net transformation (weight) implied by these phases. First, we define two procedures:

$$\begin{aligned}
\langle p_1, \gamma_1\ \gamma_2\ \gamma_3 \cdots \gamma_n \rangle &\Rightarrow^{\sigma_1} \quad \langle p_2, \gamma_2\ \gamma_3 \cdots \gamma_{k+1}\ \gamma_{k+2} \cdots \gamma_n \rangle \\
&\Rightarrow^{\sigma_2} \quad \langle p_3, \gamma_3 \cdots \gamma_{k+1}\ \gamma_{k+2} \cdots \gamma_n \rangle \\
&\cdots \\
&\Rightarrow^{\sigma_k} \quad \langle p_{k+1}, \gamma_{k+1}\ \gamma_{k+2} \cdots \gamma_n \rangle \\
&\Rightarrow^{\sigma_{k+1}} \langle p_{k+2}, u_1\ u_2 \cdots u_j\ \gamma_{k+2} \cdots \gamma_n \rangle
\end{aligned}$$

Fig. 1. A path in the PDS's transition relation; $u_i \in \Gamma, j \geq 1, k < n$.

1. $pop : P \times \Gamma \times P \to D$ is defined as follows:
$$pop(p, \gamma, p') = \bigoplus \{v(\sigma) \mid \langle p, \gamma \rangle \Rightarrow^{\sigma} \langle p', \varepsilon \rangle\}$$
2. $grow : P \times \Gamma \to ((P \times \Gamma^+) \to D)$ is defined as follows:
$$grow(p, \gamma)(p', u) = \bigoplus \{v(\sigma) \mid \langle p, \gamma \rangle \Rightarrow^{\sigma} \langle p', u \rangle\}$$

Note that $grow(p, \gamma) = poststar(\langle p, \gamma \rangle)$, where the latter is interpreted as a function from configurations to weights. The following lemmas give efficient algorithms for computing the above procedures. Proofs are given in [16].

Lemma 3. *Let $\mathcal{A} = (P, \Gamma, \emptyset, P, P)$ be a $\mathcal{P}$-automaton that represents the set of configurations $C = \{\langle p, \varepsilon \rangle \mid p \in P\}$. Let $\mathcal{A}_{pop}$ be the forward weighted-automaton obtained by running prestar on $\mathcal{A}$. Then $pop(p, \gamma, p')$ is the weight on the transition (p, γ, p') in $\mathcal{A}_{pop}$. We can generate $\mathcal{A}_{pop}$ in time $O_s(|P|^2|\Delta|H)$, and it has at most $|P|$ states.*

Lemma 4. *Let $\mathcal{A}_F = (Q, \Gamma, \rightarrow, P, F)$ be a $\mathcal{P}$-automaton, where $Q = P \cup \{q_{p,\gamma} \mid p \in P, \gamma \in \Gamma\}$ and $p \xrightarrow{\gamma} q_{p,\gamma}$ for each $p \in P, \gamma \in \Gamma$. Then $\mathcal{A}_{\{q_{p,\gamma}\}}$ represents the configuration $\langle p, \gamma \rangle$. Let $\mathcal{A}$ be this automaton where we leave the set of final states undefined. Let $\mathcal{A}_{grow}$ be the backward weighted-automaton obtained from running poststar on $\mathcal{A}$ (poststar does not need to know the final states). If we restrict the final states in $\mathcal{A}_{grow}$ to be just $q_{p,\gamma}$,we obtain a backward weighted-automaton $\mathcal{A}_{p,\gamma} = poststar(\langle p, \gamma \rangle) = grow(p, \gamma)$. We can compute $\mathcal{A}_{grow}$ in time $O_s(|P||\Delta|(|P||\Gamma| + |\Delta|)H)$, and it has at most $|P||\Gamma| + |\Delta|$ states.*

The advantage of the construction presented in Lemma 4 is that it just requires a single *poststar* query to compute all of the $\mathcal{A}_{p,\gamma}$, instead of one query for each $p \in P$ and $\gamma \in \Gamma$. Because the standard *poststar* algorithm builds an automaton that is larger than the input automaton (Lemma 2), $\mathcal{A}_{grow}$ has many fewer states than those in all of the individual $\mathcal{A}_{p,\gamma}$ automata put together.

The idea behind our approach is to use $\mathcal{A}_{pop}$ to simulate the first phase where the PDS pops off stack symbols. With reference to Fig. 1, the transducer consumes $\gamma_1 \cdots \gamma_k$ from the input tape. When the transducer (non-deterministically) decides to switch over to the growth phase, and is in state p_{k+1} in $\mathcal{A}_{pop}$ with γ_{k+1} being the next symbol in the input, it passes control to $\mathcal{A}_{p_{k+1},\gamma_{k+1}}$ to start generating the output $u_1 \cdots u_j$. Then it moves into an accept phase where it copies the untouched part of the input stack ($\gamma_{k+2} \cdots \gamma_n$) to the output.

Note that $\mathcal{A}_{pop}$ is a forward-weighted automaton, whereas $\mathcal{A}_{grow}$ is a backward-weighted automaton. Therefore, when we mix them together to build a transducer, we must allow it to switch directions for computing the weight of a path.

Consider Fig. 1; a PDS rule sequence consumes the input configuration from left to right (in the pop phase), but produces the output stack configuration u from right to left (as it pushes symbols on the stack). Because we need the transducer to output $u_1 \cdots u_j$ from left to right, we need to switch directions for computing the weight of a path. For this, we define *partitioned* transducers.

Definition 9. *A **partitioned weighted finite-state transducer** τ is a tuple $(Q, \{Q_i\}_{i=1}^2, \mathcal{S}, \Sigma_i, \Sigma_o, \lambda, I, F)$ where Q is a finite set of states, $\{Q_1, Q_2\}$ is a partition of Q, $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ is a bounded idempotent semiring, Σ_i and Σ_o are input and output alphabets, $\lambda \subseteq Q \times D \times (\Sigma_i \cup \{\varepsilon\}) \times (\Sigma_o \cup \{\varepsilon\}) \times Q$ is the transition relation, $I \subseteq Q_1$ is the set of initial states, and $F \subseteq Q_2$ is the set of final states. We restrict the transitions that cross the state partition: if $(q, w, a, b, q') \in \lambda$ and $q \in Q_l, q' \in Q_k$ and $l \neq k$, then $l = 1, k = 2$ and $w = \overline{1}$. Given a state $q \in I$, the transducer accepts a string $\sigma_i \in \Sigma_i^*$ with output $\sigma_o \in \Sigma_o^*$ if there is a path from state q to a final state that takes input σ_i and outputs σ_o.*

For a path η that goes through states $q_1, \cdots, q_m$, such that the weight of the i^{th} transition is w_i, and all states q_i are in Q_j for some j, then the weight of this path $v(\eta)$ is $w_1 \otimes w_2 \otimes \cdots \otimes w_m$ if $j = 1$ and $w_m \otimes w_{m-1} \otimes \cdots \otimes w_1$ if $j = 2$, i.e., the state partition determines the direction in which we perform extend. For a path η that crosses partitions, i.e., $\eta = \eta_1 \eta_2$ such that each η_j is a path entirely inside Q_j, then $v(\eta) = v(\eta_1) \otimes v(\eta_2)$.

In this paper, we refer to partitioned weighted transducers as weighted transducers, or simply transducers when there is no possibility of confusion. Note that when the extend operator is commutative, as in the case of the Boolean semiring used for encoding PDSs as WPDSs, the partitioning is unnecessary.

Theorem 1. *We can compute a transducer $\tau_{\mathcal{W}}$ such that if $\tau_{\mathcal{W}}$ is given input $(p\ u)$, $p \in P, u \in \Gamma^*$, then the combine over the values of all paths in $\tau_{\mathcal{W}}$ that output the string $(p'\ u')$ is precisely* $\mathrm{MOVP}(\{\langle p, u\rangle\}, \{\langle p', u'\rangle\})$. *Moreover, this transducer can be constructed in time $O_s(|P||\Delta|(|P||\Gamma| + |\Delta|)H)$, has at most $|P|^2|\Gamma| + |P||\Delta|$ states and at most $|P|^2|\Delta|^2$ transitions.*

Usually the WPDSs used for modeling programs have $|P| = 1$ and $|\Gamma| < |\Delta|$. In that case, constructing a transducer has similar complexity and size as running a single *poststar* query; see [16].

6 Composing Weighted Transducers

Composition of unweighted transducers is straightforward, but this is not the case with weighted transducers. The requirement here is to take two weighted transducers, say τ_1 and τ_2, and create another one, say τ_3, such that $\mathcal{L}(\tau_3) = \mathcal{L}(\tau_1); \mathcal{L}(\tau_2)$. The difficulty is that this requires composition of weighted languages (see Defn. 8). We begin with a slightly simpler problem on weighted automata. The machinery that we develop for this problem will be used for composing weighted transducers.

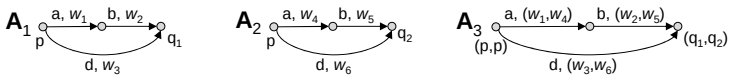

Fig. 2. Forward-weighted automata with final states q_1, q_2, and (q_1, q_2), respectively.

6.1 The Sequential Product of Two Weighted Automata

Given forward-weighted automata $\mathcal{A}_1$ and $\mathcal{A}_2$, we define their sequential product as another weighted automaton $\mathcal{A}_3$ such that for any configuration c, $\mathcal{A}_3(c) = \mathcal{A}_1(c) \otimes \mathcal{A}_2(c)$. More generally, we want the following identity for any regular set of configurations C: $\mathcal{A}_3(C) = \bigoplus\{\mathcal{A}_3(c) \mid c \in C\} = \bigoplus\{\mathcal{A}_1(c) \otimes \mathcal{A}_2(c) \mid c \in C\}$. (In this section, we assume that configurations consist of just the stack and $|P| = 1$.) This problem is the special case of transducer composition when a transducer only has transitions of the form (γ/γ). For the Boolean weight domain, it reduces to unweighted automaton intersection.

To take the sequential product of weighted automata, we start with the algorithm for intersecting unweighted automata. This is done by taking transitions (q_1, γ, q_2) and (q_1', γ, q_2') in the respective automata to produce $((q_1, q_1'), \gamma, (q_2, q_2'))$ in the new automaton. We would like to do the same with weighted transitions: given weights of the matching transitions, we want to compute a weight for the created transition. In Fig. 2, intersecting automata $\mathcal{A}_1$ and $\mathcal{A}_2$ produces $\mathcal{A}_3$ (ignore the weights for now). Automaton $\mathcal{A}_3$ should accept (`a b`) with weight $\mathcal{A}_1(\texttt{a b}) \otimes \mathcal{A}_2(\texttt{a b}) = w_1 \otimes w_2 \otimes w_4 \otimes w_5$.

One way of achieving this is to pair the weights while intersecting (as shown in $\mathcal{A}_3$ in Fig. 2). Matching the transitions with weights w_1 and w_4 produces a transition with weight (w_1, w_4). For reading off weights, we need to define operations on paired weights. Define extend on pairs ($\otimes_p$) to be componentwise extend ($\otimes$). Then $\mathcal{A}_3(\texttt{a b}) = (w_1, w_4) \otimes_p (w_2, w_5) = (w_1 \otimes w_2, w_4 \otimes w_5)$. Taking an extend of the two components produces the desired answer. Thus, this $\mathcal{A}_3$ together with a read out operation in the end (that maps a weight pair to a weight) is a first attempt at constructing the sequential product of $\mathcal{A}_1$ and $\mathcal{A}_2$.

Because the number of accepting paths in an automaton may be infinite, one also needs a combine ($\oplus_p$) on paired weights. The natural attempt is to define it componentwise. However, this is not precise. For example, if $C = \{c_1, c_2\}$ then $\mathcal{A}_3(C)$ should be $(\mathcal{A}_1(c_1) \otimes \mathcal{A}_2(c_1)) \oplus (\mathcal{A}_1(c_2) \otimes \mathcal{A}_2(c_2))$. However, using componentwise combine, we would get $\mathcal{A}_3(C) = \mathcal{A}_3(c_1) \oplus_p \mathcal{A}_3(c_2) = (\mathcal{A}_1(c_1) \oplus \mathcal{A}_1(c_2), \mathcal{A}_2(c_1) \oplus \mathcal{A}_2(c_2))$. Applying the read-out operation (extend of the components) gives four terms $\bigoplus\{(\mathcal{A}_1(c_i) \otimes \mathcal{A}_2(c_j)) \mid 1 \le i, j \le 2\}$, which includes cross terms like $\mathcal{A}_1(c_1) \otimes \mathcal{A}_2(c_2)$. The same problem arises also for a single configuration c if $\mathcal{A}_3$ has multiple accepting paths for it.

Under certain circumstances there is an alternative to pairing that lets us compute precisely the desired sequential product of weighted automata:

Definition 10. *The n^{th} **sequentializable tensor product** (n-STP) of a weight domain $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ is defined as another weight domain $\mathcal{S}_t =$*

$(D_t, \oplus_t, \otimes_t, \overline{0}_t, \overline{1}_t)$ *with operations* $\odot : D^n \to D_t$ *(called the tensor operation) and* $DeTensor : D_t \to D$ *such that for all* $w_j, w'_j \in D$ *and* $t_1, t_2 \in D_t$,

1. $\odot(w_1, w_2, \cdots, w_n) \otimes_t \odot(w'_1, w'_2, \cdots, w'_n) = \odot(w_1 \otimes w'_1, w_2 \otimes w'_2, \cdots, w_n \otimes w'_n)$
2. $DeTensor(\odot(w_1, w_2, \cdots, w_n)) = (w_1 \otimes w_2 \otimes \cdots \otimes w_n)$ *and*
3. $DeTensor(t_1 \oplus_t t_2) = DeTensor(t_1) \oplus DeTensor(t_2)$.

When $n = 2$, we write the tensor operator as an infix operator. Note that because of the first condition in the above definition, $\overline{1}_t = \odot(\overline{1}, \cdots, \overline{1})$ and $\overline{0}_t = \odot(\overline{0}, \cdots, \overline{0})$. Intuitively, one may think of the tensor product of i weights as a kind of generalized i-tuple of those weights. The first condition above implies that extend of tensor products must be carried out componentwise. The *DeTensor* operation is the "read-out" operation that puts together a tensor product by taking extend of its components. The third condition distinguishes the tensor product from a simple tupling operation. It enforces that the *DeTensor* operation distributes over the combine of the tensored domain, which pairing does not satisfy.

If a 2-STP exists for a weight domain, then we can take the product of weighted automata for that domain: if $\mathcal{A}_1$ and $\mathcal{A}_2$ are the two input automata, then for each transition (p_1, γ, q_1) with weight w_1 in $\mathcal{A}_1$, and transition (p_2, γ, q_2) with weight w_2 in $\mathcal{A}_2$, add the transition $((p_1, p_2), \gamma, (q_1, q_2))$ with weight $(w_1 \odot w_2)$ to $\mathcal{A}_3$. The resulting automaton satisfies the property: $DeTensor(\mathcal{A}_3(c)) = \mathcal{A}_1(c) \otimes \mathcal{A}_2(c)$, and more generally, $DeTensor(\mathcal{A}_3(C)) = \bigoplus\{\mathcal{A}_1(c) \otimes \mathcal{A}_2(c) \mid c \in C\}$. (A proof is given in [16].) Thus, with the application of the *DeTensor* operation, $\mathcal{A}_3$ behaves like the desired automaton for the product of $\mathcal{A}_1$ and $\mathcal{A}_2$. A similar construction and proof hold for taking the product of n automata at the same time, when an n-STP exists.

Before generalizing to composition of transducers, we show that n-STP exists, for all n, for a class of weight domains. This class includes the one needed to perform affine-relation analysis [16].

6.2 Sequentializable Tensor Product

We say that a weight domain is commutative if its extend is commutative. STP is easy to construct for commutative domains (tensor is extend, and *DeTensor* is identity), but such domains are not useful for encoding abstractions for CBA. Under a commutative extend, interference from other threads can have no effect on the execution of a thread. However, such domains still play an important role in constructing STPs. We show that STPs can be constructed for *matrix domains* built on top of a commutative domain.

Definition 11. *Let* $\mathcal{S}_c = (D_c, \oplus_c, \otimes_c, \overline{0}_c, \overline{1}_c)$ *be a commutative weight domain. Then a* ***matrix weight domain*** *on* $\mathcal{S}_c$ *of order* n *is a weight domain* $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ *such that* D *is the set of* $n \times n$ *matrices with elements from* D_c*;* $\oplus$ *is element-wise* $\oplus_c$*;* $\otimes$ *is matrix multiplication;* $\overline{0}$ *is the matrix in which all elements are* $\overline{0}_c$*;* $\overline{1}$ *is the matrix with* $\overline{1}_c$ *on the diagonal and* $\overline{0}_c$ *elsewhere.*

$\mathcal{S}$ is a bounded idempotent semiring even if $\mathcal{S}_c$ is not commutative.

The advantage of looking at weights as matrices is that it gives us essential structure to manipulate for constructing the STP. We need the following operation on matrices: the *Kronecker product* [29] of two matrices A and B, of sizes $n_1 \times n_2$ and $n_3 \times n_4$, respectively, is the matrix C of size $(n_1\ n_3) \times (n_2\ n_4)$ such that $C(i,j) = A(i \operatorname{div} n_3, j \operatorname{div} n_4) \otimes B(i \mod n_3, j \mod n_4)$, where matrix indices start from zero. It is much easier to understand this definition pictorially (writing $A(i,j)$ as a_{ij}):

$$C = \begin{pmatrix} a_{11}B & \cdots & a_{1n_2}B \\ \vdots & \ddots & \vdots \\ a_{n_11}B & \cdots & a_{n_1n_2}B \end{pmatrix}$$

The Kronecker product, written as $\odot$, is an associative operation. Moreover, for matrices A, B, C, D with elements that have commutative multiplication, $(A \odot B) \otimes (C \odot D) = (A \otimes C) \odot (B \otimes D)$ [29].

Note that the Kronecker product has all pairwise products of elements from the original matrices. One can come up with *projection* matrices p_i (with just $\overline{1}$ and $\overline{0}$ entries) such that $p_i \otimes m \otimes p_j$ selects the (i,j) entry of m (zeros out other entries). Using these matrices in conjunction with *permutation* matrices (that can permute rows and columns and change the size of a matrix), one can compute the product of two matrices from their Kronecker product: there are fixed matrices e_i, e_j and an expression $\theta_m = \bigoplus_{i,j}(e_i \otimes m \otimes e_j)$, such that $\theta_{m_1 \odot m_2} = m_1 \otimes m_2$. This can be generalized to multiple matrices to get an expression θ_m of the same form as above, such that $\theta_{m_1 \odot \cdots \odot m_n} = m_1 \otimes \cdots \otimes m_n$. The advantage of having an expression of this form is that $\theta_{m_1 \oplus m_2} = \theta_{m_1} \oplus \theta_{m_2}$ (because matrix multiplication distributes over their addition, or combine).

Theorem 2. *A n-STP exists on matrix domains for all n. If $\mathcal{S}$ is a matrix domain of order r, then its n-STP is a matrix domain of order r^n with the following operations: the tensor product of weights is defined as their Kronecker product, and the DeTensor operation is defined as $\lambda m.\theta_m$.*

The necessary properties for the tensor operation follow from those for Kronecker product (this is where we need commutativity of the underlying semiring) and the expression θ_m. This also implies that the tensor operation is associative and one can build weights in the n^{th} STP from a weight in the $(n-1)^{th}$ STP and the original matrix weight domain by taking the Kronecker product. It follows that the sequential product of n automata can be built from that of the first $(n-1)$ automata and the last automaton. The same holds for composing n transducers. Therefore, for CBA, the context-bound can be increased incrementally, and the transducer constructed for $(\Rightarrow^c)^k$ can be used to construct one for $(\Rightarrow^c)^{k+1}$.

6.3 Composing Transducers

Unweighted transducer composition proceeds in a similar fashion to automaton intersection: for transitions $(q_1, \gamma_1/\gamma_2, q_2)$ and $(q_1', \gamma_2/\gamma_3, q_2')$ in the respective transducers, add $((q_1, q_1'), \gamma_1/\gamma_3, (q_2, q_2'))$ to the new transducer.

If our weighted transducers were unidirectional (completely forwards or completely backwards) then composing them would be the same as taking the product of weighted automata: the weights on matching transitions would get tensored together. However, our transducers are partitioned, and have both a forwards component and a backwards component. To handle the partitioning, we need additional operations on weights.

Definition 12. *Let $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ be a weight domain. Then a **transpose** operation on this domain is defined as $(.)^T : D \to D$ such that for all $w_1, w_2 \in D$, $w_1^T \otimes w_2^T = (w_2 \otimes w_1)^T$ and it is its self-inverse: $(w_1^T)^T = w_1$. An n-**transposable STP** (TSTP) on $\mathcal{S}$ is defined as an n-STP along with another de-tensor operation: TDeTensor : $D^n \to D$ such that TDeTensor$(\odot(w_1, w_2, \cdots, w_n)) = w_1 \otimes w_2^T \otimes w_3 \otimes w_4^T \otimes \cdots w'_n$, where $w'_n = w_n$ if n is odd and w_n^T if n is even.*

TSTPs always exist for matrix domains: the transpose operation is the matrix-transpose operation, and the *TDeTensor* operation can be defined using an expression similar to that for *DeTensor*. We can use TSTPs to remove the partitioning. Let τ be a partitioned weighted transducer on $\mathcal{S}$, for which a transpose exists, as well as a 2-TSTP. The partitioning on the states of τ naturally defines a partitioning on its transitions as well (a transition is said to belong to the partition of its source state). Replace weights w_1 in the first (forwards) partition with $(w_1 \odot \overline{1})$, and weights w_2 in the second (backwards) partition with $(\overline{1} \odot w_2^T)$. This gives a completely forwards transducer τ' (without any partitioning). The invariant is that for any sets of configurations S and T, $\tau(S, T)$, which is the combine over all weights with which the transducer accepts $(s, t), s \in S, t \in T$, equals *TDeTensor*$(\tau'(S, T))$.

This can be extended to compose partitioned weighted transducers. Composing n transducers requires a $2n$-TSTP: each transducer is converted to a non-partitioned one over the 2-TSTP domain; input/output labels are matched just as for unweighted transducers; and the weights are tensored together.

Theorem 3. *Given n weighted transducers $\tau_1, \cdots, \tau_n$ on a weight domain with $2n$-TSTP, the above construction produces a weighted transducer τ such that for any sets of configurations S and T, TDeTensor$(\tau(S, T)) = R(S, T)$, where R is the weighted composition of $\mathcal{L}(\tau_1), \cdots, \mathcal{L}(\tau_n)$.*

By composing the weighted languages of transducers, we can construct a weighted transducer τ for $(\Rightarrow^c)^{k+1}$. If automaton $\mathcal{A}_S$ represents the set of starting states of a program, $\tau(\mathcal{A}_S)$ provides a weighted automaton $\mathcal{A}$ describing all reachable states (under the context bound), i.e., the weight $\mathcal{A}(t)$ gives the net transformation in data state in going from S to t ($\overline{0}$ if t is not reachable).

7 Related Work

Reachability analysis of concurrent recursive programs has also been considered in [4, 22, 9]. These consider the problem by computing overapproximations of the

execution paths of the program, whereas here we compute underapproximations of the reachable configurations. Analysis under restricted communication policies (in contrast to shared memory) has also been considered [6, 14]. The basic technique of Qadeer and Rehof has been generalized to handle more abstractions in [5, 3], however, these also require enumeration of states at a context switch, and cannot handle infinite-state abstractions like affine-relation analysis.

The goal of partial-order reduction techniques [12] for concurrent programs is to avoid explicit enumeration of all interleavings. Our work is in similar spirit, however, we use symbolic techniques to avoid explicitly considering all interleavings. In the QR algorithm, only the stack was kept symbolic using automata, and we extended that approach to keep both stack and data symbolic.

Constructing transducers. As mentioned in the introduction, a transducer construction for PDSs was given earlier by Caucal [8]. However, the construction was given for prefix-rewriting systems in general and is not accompanied by a complexity result, except for the fact that it runs in polynomial time. Our construction for PDSs, obtained as a special case of the construction given in §5, is quite efficient. The technique, however, seems to be related. Caucal constructed the transducer by exploiting the fact that the language of the transducer is a union of the relations $(pre^*(\langle p, \gamma \rangle), post^*(\langle p, \gamma \rangle))$ for all $p \in P$ and $\gamma \in \Gamma$, with an identity relation appended onto them to accept the untouched part of the stack. This is similar to our decomposition of PDS paths (see Fig. 1). Construction of a transducer for WPDSs has not been considered before. This was crucial for developing an algorithm for CBA with infinite-state data abstractions.

Composing transducers. There is a large body of work on weighted automata and weighted transducers in the speech-recognition community [17, 18]. However, the weights in their applications usually satisfy many more properties than those of a semiring, including (i) the existence of an inverse, and (ii) commutativity of extend. We refrain from making such assumptions.

The sequential product of weighted automata on semirings was also considered in [15]. However, that algorithm handles only the special case of taking one product of a forwards automaton with a backwards one. It cannot take the product of three or more automata. The techniques in this paper are for taking the product any number of times (provided STPs exist).

Tensor products have been used previously in program analysis for combining abstractions [21]. We use them in a different context and for a different purpose. In particular, previous work has used them for combining abstractions that are performed in *lock-step*; in contrast, we use them to stitch together the data state *before* a context switch with the data state *after* a context switch.

References

1. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, 2001.
2. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *CONCUR*, 1997.

3. A. Bouajjani, J. Esparza, S. Schwoon, and J. Strejcek. Reachability analysis of multithreaded software with asynchronous communication. In *FSTTCS*, 2005.
4. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *POPL*, 2003.
5. A. Bouajjani, S. Fratani, and S. Qadeer. Context-bounded analysis of multithreaded programs with dynamic linked structures. In *CAV*, 2007.
6. A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *CONCUR*, 2005.
7. J. R. Büchi. *Finite Automata, their Algebras and Grammars*. Springer-Verlag, 1988. D. Siefkes (ed.).
8. D. Caucal. On the regular structure of prefix rewriting. *TCS*, 106(1):61–86, 1992.
9. S. Chaki, E. M. Clarke, N. Kidd, T. W. Reps, and T. Touili. Verifying concurrent message-passing C programs with recursive calls. In *TACAS*, 2006.
10. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *CAV*, 2000.
11. A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *Electronic Notes in Theoretical Comp. Sci.*, 9, 1997.
12. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer-Verlag, 1996.
13. J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
14. V. Kahlon and A. Gupta. On the analysis of interacting pushdown systems. In *POPL*, 2007.
15. A. Lal, N. Kidd, T. Reps, and T. Touili. Abstract error projection. In *SAS*, 2007.
16. A. Lal, T. Touili, N. Kidd, and T. Reps. Interprocedural analysis of concurrent programs under a context bound. TR-1598, University of Wisconsin, July 2007.
17. M. Mohri, F. Pereira, and M. Riley. Weighted automata in text and speech processing. In *ECAI*, 1996.
18. M. Mohri, F. Pereira, and M. Riley. The design principles of a weighted finite-state transducer library. In *TCS*, 2000.
19. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *POPL*, 2004.
20. M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, 2007.
21. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
22. G. Patin, M. Sighireanu, and T. Touili. Spade: Verification of multithreaded dynamic and recursive programs. In *CAV*, 2007.
23. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, 2005.
24. S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In *PLDI*, 2004.
25. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. In *TOPLAS*, 2000.
26. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.
27. T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *SCP*, volume 58, 2005.
28. S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technical Univ. of Munich, Munich, Germany, July 2002.
29. Wikipedia. Kronecker product. http://en.wikipedia.org/wiki/Kronecker_product.